An Efficient Technique for Tracking Nondeterministic Execution and its Applications

E.N. Elnozahy

May 1995

CMU-CS-95-157

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This document is an updated revision of a research initiation award proposal that was submitted to and funded by the National Science Foundation in January 1994 and September 1994, respectively.

This work is supported in part by a National Science Foundation Research Initiation Award under grant number CCR-9410116.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the NSF or the U.S. Government.

Keywords: Debugging, Distributed Systems, Fault Tolerance, Heizenbugs, High-Availability, Instruction Counters, Monitoring, Nondeterminism, Operating Systems, PA-RISC, Performance, Rollback-Recovery

Abstract

This report describes a technique for using instruction counters to track nondeterminism in the execution of operating system kernels and user programs. The operating system records the number of instructions between consecutive nondeterministic events and information about their nature during normal operation. During an analysis phase, the execution is repeated under the control of a monitor, and the nondeterministic events are applied at the same instructions as during the monitored execution. We describe the application of this technique to four areas:

Performance monitoring: The technique can be used to instrument an operating system to capture long traces of memory references. Unlike current techniques, it performs the gathering in a postmortem phase and therefore has negligible effect on the computation itself during the monitoring phase. We expect trace periods that are longer than what existing techniques can capture by orders of magnitude with little or no noticeable perturbation to the monitored system itself.

Kernel Debugging: This technique can be used to repeat the execution of an operating system that precedes a crash due to a Heizenbug. This allows developers a systematic approach for getting rid of these bugs during testing.

Support for Rollback-Recovery: Systems that use checkpointing and execution replay can adopt this technique to ensure that execution replay during recovery is identical to the one before failure, despite the occurence of nondeterministic events that cannot be captured efficiently otherwise.

Software-based TMR systems: Using this technique, a TMR system based on active replication can be built out of off-the-shelf workstations connected by a general purpose network. Nondeterministic events occuring in a primary can be emulated on backup machines to ensure identical execution.

We plan to implement this technique on two architectures. The first is an HP platform based on the PA-RISC architecture which supports instruction counters in hardware. The second is a MIPS-based architecture and in which programs are processed to emulate an instruction counter in software.

1 Summary

Tools for monitoring the behavior of an operating system are invaluable in performance tuning and debugging. However, monitoring operating systems under realistic workloads is a difficult problem. The objective of this project is to develop a monitoring tool that allows the performance of the system to be studied under such workloads.

Unlike previous hardware and software monitoring tools, the new tool proposed here can collect long execution traces with very little perturbation to the actual system. The duration of the monitored execution is estimated to be orders of magnitude longer than is possible using any existing technique. The tool allows the studying of system performance under real-time workloads and network activities, both of which are very difficult to monitor using existing software techniques because of the excessive processing overhead of such methods. Implementation takes place entirely in software, and relies on the instruction counters that are becoming available in an increasing number of modern processors. Because the execution of the system is not affected by the monitoring process, it can be applied to collect very long traces while the system is in production use. This will allow more ambitious performance studies than is possible today. The monitor can also be applied for debugging purposes during the final testing phases to uncover intermittent bugs. These bugs, known also as Heizenbugs, are the most difficult to discover and are the reason for the perceived unreliability of systems software in the user community. Unfortunately, techniques for uncovering these bugs once the system is in production use do not exist. The work presented here presents a fresh approach to attack this important problem. Extensions of the ideas involved to rollback-recovery, replication, and distributed systems will also be investigated. The work has the potential to study the performance of distributed and parallel programs, which constitute the next challenging front.

2 Introduction

The increasing dependence of the society on computers and the emerging ambitious applications are imposing stricter performance and reliability requirements on operating system implementations. Recent studies, however, have shown that faster processors and networks do not necessarily lead to similar improvements in operating system performance [3, 19]. Furthermore, the expanded functionality of modern operating systems often results in complex implementations of questionable reliability [12].

Part of the problem is that while the process of measuring the performance of various components of a system in isolation is well understood, it is very difficult to comprehensively measure the performance to account for the interactions between the different components. For example, the context switching overhead in most systems is negligible if considered in isolation. Similarly, several studies indicated that the hit ratio of typical cache memory systems is in the upper 90%'s. However, when the interactions between the context switching overhead and the cache memory system are considered, it becomes clear that each of these two components severely affects the performance of the other.

Many tools have been developed to try to characterize the system's performance under a multiprogrammed, realistic workload [1, 2, 4, 11, 14, 17, 20]. A typical study using one of these tools consists of the collection of a trace for memory accesses, followed by a simulation that uses the collected trace information. Several studies used these tools to add valuable insight into the problem of designing and evaluating system software. However, all these tools have their shortcomings. First, because they collect memory references, the volume of the trace information is very large. The required storage overhead limits the duration of trace collection to a few seconds, typically four or five [20]. Furthermore, hardware monitoring techniques are expensive and inflexible. They also cannot work with the emerging new generation of processors that have on-chip caches unless they are disabled. Software techniques on the other hand are flexible, but they introduce a substantial slowdown of the monitored program (a factor of 10 to 15). This slowdown affects the accuracy of the collected information, and requires corrective measures during the simulation phase, which is not a straightforward procedure. This problem makes it very difficult to collect traces while the system is in production use. These software tools also are not suitable for collecting trace information under a real-time workload or when network activities need to be monitored.

We propose a new monitoring tool that uses a different technology from existing tools. Instead of collecting the memory references on the fly, the tool collects information that allows these references to be generated during a post-mortem run. The collected information is orders of magnitude smaller than a trace of memory references. Therefore, the proposed tool can collect traces for longer executions. In addition, the perturbations added to the system execution are very small. Thus, there is no restriction on the type of workload that can be monitored, and a realistic workload can be collected from a system while it is in production use. The new technique avoids the problems of existing tools, and allows more ambitious performance studies that are not possible or are very difficult to do in existing techniques.

The new technique proposed here works by tracking the nondeterministic events that occur during a system execution. A monitor is compiled into the kernel and receives requests for beginning or terminating the monitoring procedure. At the beginning of the monitoring process, the state of the machine is saved. The system is then allowed to resume execution. The monitor intercepts every asynchronous interrupt and saves sufficient information about it in a pre-allocated memory buffer. After the monitoring process terminates, a postmortem analyzer uses the information gathered by the monitor to perform various measurements or simulations. The analyzer restores the machine's state that was saved, and restarts execution by single stepping or executing a number of instructions defined by some user routine. The information collected during the initial run is used to apply the asynchronous interrupts at the same locations as they occurred in the initial run. The execution replay is thus identical to that of the monitored system. The techniques are implemented entirely in software, and they rely on the existence of an instruction counter in the processor architecture for efficiency. Given that an increasing number of modern processors are providing such counters, this requirement is hardly a restriction. Furthermore, instruction counters can be emulated in software for those architectures that lack this support.

During postmortem analysis, the memory references can be tracked as in conventional tools and fed to a simulator. The output of that tool, however, is not limited to generating memory references. In fact, the traced information provides a complete description of the system execution that can be used for any purpose. For instance, network activities can be monitored with accuracy.

The applications of the proposed tool are not restricted to performance measurements and evaluation. It can also be applied to uncover intermittent bug during the final testing phases of the operating system. Commonly, a beta-version of the operating system is made available for initial testing. The bug reports collected are analyzed and appropriate fixes are incorporated into the final version. However, some bugs are very difficult to reproduce. Such bugs, also known as Heizenbugs, are a result of a sequence of nondeterministic events that are difficult to reproduce because insufficient information is usually available for analysis. The final release of an operating system is typically shipped with many of these lurking bugs.

Since the monitoring tool depends in its operation on tracking the nondeterministic events, it is ideally positioned to provide some discipline into the final testing phases of the pre-release code. The collected information is used during the postmortem analysis to reproduce the Heizenbugs. The information can be then fed to a debugger that can analyze the situation given the trace of events. Of course, special provision must be taken to protect the monitor code and collected data from being corrupted because of system bugs.

The technique proposed also has applications in rollback-recovery and can be extended to measure the performance of a distributed system. These issues are discussed in the remainder of the report. The basic idea is described in some detail in Section 3. The postmortem analysis is treated in Section 4. The proposed work is compared to other work in the field in Section 5. Finally, the work plan is discussed in Section 6 and the impact of the proposed research is discussed in section 7.

3 The Operating System Monitor

3.1 Background

An increasing number of modern RISC processors include support for debugging in the form of instruction or breakpoint counters. Examples include the DEC Alpha, Intel Pentium, and HP PA-RISC, among others. This support allows the operating system to control the processor to generate an interrupt after executing a specified number of instructions. The operating system typically sets a special hardware register with the number of instructions to be executed. This counter is decremented automatically by one each time the processor executes one instruction. The processor generates an interrupt when the counter reaches zero. The hardware implementation ensures that the value maintained in the register corresponds to the sequential execution of the particular program, even in the presence of instruction pipelining and out of order scheduling.

The techniques presented in this report rely on the existence of such debugging support, which may take several different forms. To make the discussion concrete, the presentation will use the HP PA-RISC architecture as an example [13]. This architecture provides a *recovery counter* that can be modified under operating system control. The recovery counter is decremented by one after the execution of each instruction, and the processor generates a *recovery counter interrupt* when the counter reaches zero. With minor modifications, the proposed techniques can easily be implemented on different architectures that provide similar support.

3.2 Nondeterminism

The execution of a computer is a sequence of state transitions. Without external input and interrupts, the state transitions are deterministic. In other words, the computer system ends in the same state if it starts from the same initial state and executes the same number of instructions in each run. Nondeterminism occurs, however, if the processor reads data from input devices or receives asynchronous interrupts. When such events occur, the future state transitions generally depend on the data read, or on the exact time during which an asynchronous interrupt occurs. Thus, the execution of a program is a function of the initial state, input data, and the particular interleaving of asynchronous events that occur during the execution.

Two executions of the same program that start from the same initial state can be made identical if both runs experience the same interleaving of asynchronous events and receive the same input values. The basic thesis of this report is that an efficient monitoring tool for operating systems can be built by tracking the nondeterministic events during an actual run, and applying them during a postmortem run that starts from the same state as the actual one. This idea has applications in performance analysis and in uncovering intermittent bugs. Unlike previous tools, the proposed one can be applied during *production-use* with very little execution perturbation. Therefore, an accurate description of the system operation can be provided in a real setting.

3.3 An Efficient Monitoring Tool

The proposed monitor is implemented by a collection of routines that are added to the operating system. Monitoring starts by sending a request to the kernel. The monitor saves the state of the machine and allows the execution of the system to resume. The monitor *conceptually* records information about all input data and asynchronous interrupts in a pre-allocated memory buffer. Monitoring terminates by sending a request to the kernel or when the monitor exhausts the resources allocated for recording the nondeterministic events. The initial state of the machine is then reloaded, and the system re-executes under the control of a postmortem analyzer. The analyzer ensures that the postmortem execution will perceive the same input data and the same interleaving of interrupts that were collected during the monitored execution. User routines can be called during this postmortem run to perform various measurements, to collect statistics, to generate memory reference traces, to perform various simulations using the workload, or to verify assertions about the execution.

The key to efficiency is that the collection of information during the actual run should be done with very little impact on the execution of the monitored program. Furthermore, the storage overhead should be low to allow the monitoring to extend over a reasonably long period. The monitor adds an *event descriptor* to a pre-allocated memory buffer each time the monitored program reads input or when an asynchronous interrupt occurs. The descriptor contains sufficient information about the event such that it can be re-applied during postmortem execution to produce the same effect. Note that traps to the kernel by user programs, page faults, TLB miss interrupts[1] do not constitute nondeterministic events, and therefore are not recorded. Such events will be regenerated during postmortem execution. Moreover, in systems where reclamation of memory pages or TLB entries follows a random policy, it is sufficient to record the value

[1]Assuming the TLB miss handler is in software.

returned by the random number generator, assuming it is an external one.[2]

3.3.1 Collecting Information about Interrupts

The monitor uses the recovery counter supplied by the processor to record the number of instructions between every two consecutive interrupts. At the beginning of the run, the monitor sets the recovery counter to its maximum value. When an interrupt occurs, the monitor records in an event descriptor the type of the interrupt and the number of instructions that were executed since the last interrupt, as computed by the recovery counter. The monitor then resets the recovery counter to the maximum value. On an architecture like the PA-RISC, these operations add only eight instructions to the interrupt handler. The monitor may also add to the event descriptor the reading of the high precision clock, if desirable. The size of the event descriptor is four bytes if no time information is recorded. Recording time information may add two to four bytes according to the resolution desired, if at least one interrupt occurs each 50 milliseconds.

During postmortem execution, the exact locations of interrupts are available by inspecting the event descriptors. Since execution is controlled, an interrupt can be applied to the system after executing the number of instructions that were executed since the last interrupt in the original run.

3.3.2 Collecting Information about Input Data

For data input events, an event descriptor contains a field that identifies the device and a field showing the time at which the event occurs. The descriptor also contains additional fields that depend on the type of the event and contain the necessary information to replay it during postmortem analysis. Typical examples of the information that must be stored for such events follow:

- For the keyboard, the event descriptor must contain the values in the particular I/O registers that identify the key depressed.
- For a serial device, the event descriptor must contain the value of the character read.
- For the timer, the event descriptor must contain the value read from the timer.
- For the mouse, the event descriptor must contain the values of the particular I/O registers that show the x and y coordinates, and whether a mouse button was clicked.

The devices described so far work in character mode, and therefore the corresponding event descriptors would typically not require more than a dozen

[2]The system typically generates random values by reading the clock. These actions are recorded as part of the monitoring process.

of bytes each. Handling other devices that operate in character mode or exchange control information with the operating system should be similar. Care must be taken for input devices that operate in block mode, such as the disk or the network. Simply copying the data to the event descriptor requires processing overhead that would certainly perturb the monitored system and affect the accuracy of the collected data. Furthermore, storing large data blocks in the monitor's buffer would exhaust it quickly, cutting short the length of the monitored execution.

For disk input, the event descriptor contains only the block number and the control information that are returned by the device. The input data can be generated during postmortem execution by including the disk image in the initial state that is stored before monitoring starts. Thus, the local disk is mirrored on some external storage device. This operation can be done efficiently by backing up the disk to another machine on the network and only storing the modified files since the last backup before starting the monitoring process. The advantage of this technique is that it includes the actual effects of disk I/O into the collected information, providing a high degree of accuracy. Alternatively, the file system may be modified to provide versioning. While monitoring is active, file modifications take place on a different version than the one that existed before monitoring started. The advantage of this technique is that it does not incur a large storage overhead, but it requires modification to the file system and may not represent the actual performance or behavior of the monitored file system. In the prototype implementation, the first approach will be followed.

Network input presents a different problem since it cannot be generated from the initial state. To solve this problem, an adjacent machine on the network is configured to listen to the network and record the packets directed to the monitored machine. These packets are made available during postmortem execution. If more than one machine is monitored over the network, only one machine is configured to snoop over the network and record all the desired packets. The event descriptor thus can contain only sufficient information to identify the packet during postmortem operation. Such information includes the source address of the packet, its CRC, protocol type, and size. The critical issue here is that there should be a one-to-one mapping between the information stored and the actual packet. Experimentation will show whether the proposed fields are sufficient to uniquely identify the packet, or whether additional information is required.

Generally, the snooping machine may collect packets that the monitored machine may miss because of buffer overflow and overload. These additional packets should not be a source of concern, as long as they are not causing any ambiguity in identification. They could also be a good source of information when studying the networking performance of the monitored machine. On the other hand, the snooping machine may miss a packet that is captured by the monitored machine. In such a case, the entire monitoring process fails. Experience shows this case can be made virtually impossible to occur by properly configuring the snooping machine with sufficient buffering and dedicating it to the measurement process [10].

3.3.3 Optimizations

In many systems, reading input data is often a result of an interrupt. Examples include keyboard, network, and disk interrupts, among many others. In such cases, the event descriptors for the interrupt and the data read can be combined in one descriptor for efficiency.

3.4 Projected Performance

The first aspect of the monitor's performance is the processing overhead it incurs in collecting the information. This overhead controls the amount of perturbation that would affect the accuracy of the results. The processing overhead of recording event descriptors is small. For an interrupt, the manipulation of the recovery counter and the construction of the event descriptor can be done in about eight to twelve instructions.[3] For data input, the manipulation required can be done in less than about 30 instructions at worst. The corresponding interrupt handlers are often hundreds or thousands of instructions long. Therefore, the projected effect of monitoring on the system's execution is small. Thus, the proposed technique allows the collection of accurate execution traces with realistic workloads. Furthermore, it allows the studying of the effects of real-time input such as the network input, which cannot be monitored in a system that imposes a large processing overhead.

The second aspect of the monitor's performance is the memory storage overhead required for the event descriptors. This overhead affects the duration of the monitoring process. Assuming a system that receives a high rate of 5000 interrupts per second, the required memory buffer storage is about 100 Kilobytes, assuming 20 bytes per event descriptor at worst. To put this figure into perspective, consider a hardware monitor that collects a memory reference trace [11], or alternatively a program that was massaged by a trace generation tool such as *epoxie* [16]. Assuming the machine generates a modest 10 million memory references per second, the required storage would be in the order of 40 Megabytes, or more than four hundred times the space required by our proposed monitor.

Note that the storage required by the network snooper is not a critical issue. Secondary storage can be used since the resulting overhead in processing does not affect in any way the collection of information on the monitored machine.

3.5 Other Implementation Issues

The particular details of an implementation depend on the architecture on which the monitor runs. Following are some implementation issues and limitations of

[3]Optional inclusion of a time reading causes the variability.

the proposed technique that would be common on many architectures.

3.5.1 Competition for the Recovery Counter

The recovery counter was not provided by the architecture designers to support our monitoring tool! Instead, programs such as debuggers may use the counter to establish breakpoints, etc... In such cases, the monitor must cooperate with these user processes. Fortunately, user processes must trap to the kernel in order to manipulate the recovery counter, and therefore the modification required is very simple. Instead of setting the counter to its maximum value, the monitor uses the value used by the user program. The monitor can use a set of auxiliary counters that can record the number of instructions between interrupts as described above. When a recovery counter interrupt occurs, the monitor simply updates the auxiliary counters. The occurrence of this interrupt is obviously not considered a nondeterministic event.

3.5.2 User Space Mapped Input

Some systems map some input devices into the user address space. The most common example is to map the hardware clock into the addressable space of user programs. If the behavior of the system will be affected by the input values processed by the user programs, such a facility must be disabled for the monitoring process to function correctly.

3.5.3 Nested Interrupts

Reading and updating the recovery counter must be performed while interrupts are masked. Otherwise, updating the counter is not atomic and incorrect values may be recorded.

3.5.4 Non Broadcast Networks

The proposed technique assumes that network packets can be read by other machines as they travel over the wire. Snooping is straightforward on broadcast networks such as Ethernet or token ring networks. On point-to-point networks, special provisions must be made to reroute the packets intended for the monitored machine to the snooping machine. Unfortunately, at the time this report is being written there does not exist a final standard for point to point networks, such as ATM. Therefore, it is not clear whether this issue is going to be a problem or not.

4 The Postmortem Analyzer

The information collected by the monitor provides an accurate account of the execution of the machine, ignoring the slight perturbation due to the event collection routines. This information is used to re-execute the run under the control of a postmortem analyzer. The analyzer consists of a few routines added to the kernel, and a number of user routines. The kernel routines are responsible for reinstalling the state of the machine, re-executing the run and replaying input data and interrupts appropriately, and calling user routines to perform necessary measurements. The postmortem analyzer may execute the run in a single-step mode, calling a user routine after executing every instruction. This arrangement is desirable for instance to implement an instruction-level simulation of the machine, or to use the monitor to debug the kernel as described in Section 4.2. Alternatively, the user routines may specify a number of instructions to be executed before being called. The relative speed of the postmortem analysis will generally depend on the processing requirements of the user routines. These routines can be used for different purposes. In this report, we consider four such applications.

4.1 Observing System Performance

The main application of the proposed technique will be the analysis of the system performance under *actual* workloads. Postmortem execution will give an accurate account of the overhead of context switching and its effect on the cache performance. It will also give an accurate account of the performance of the memory system, including the performance of the cache and TLB. The network traffic and its effect on the system performance will also be studied, in addition to its interactions with the memory system. The postmortem analyzer can also be used to collect the memory references of the monitored program and make it available to the research community at large.

Given the projected performance described in Section 3.4, it is not unreasonable to expect that about an hour worth of execution can be captured and replayed. A 64 Megabyte noncacheable memory will be necessary to store the events while the execution is monitored. The memory will not be cached in order not to perturb the execution of the monitored program. No performance problem is likely, since this memory will be always written to during monitoring and write buffers common in modern architectures would mask the latency of writing such information.

Such a long duration is, to the best of my knowledge, unprecedented. The proposed technique can also monitor real-time workloads because of the efficiency in data collection. Furthermore, it can be used to study the network behavior under actual workloads. These issues were difficult to study with previous monitors.

4.2 Extensions

4.2.1 Intermittent Bugs

The monitor can be used to track intermittent bugs in a production operating system. These bugs are often referred to as Heizenbugs, since they are difficult to reproduce and generally depend on a combination of asynchronous events occurring in a way that is difficult to anticipate or understand [12]. Such bugs manifest themselves by crashing the system or forcing the operator to reboot it. These software "failures" are supposed to be caught during what is commonly called "beta-testing." This testing refers to the phase where the company makes a pre-release of its operating system available to a limited user community to "try the system out" and produce bug reports. Such reports are analyzed and it is up to the implementation team to reason about the probable causes. The results are not always satisfactory, as it is often the case that the final release of the operating system is not of high quality.

The monitoring process described in this report can be used during beta-testing to provide an accurate account of the events that lead to the failure. It will help cut short the guessing involved and impose some discipline on the process. In such a case, the buffer used to collect the event descriptor must be set non-writeable, to prevent bugs in the operating system from modifying it inadvertently. The monitor would manipulate the protection on the buffer when it writes the information. Furthermore, the locations in the system's code that access the recovery counter must be carefully analyzed to ensure that the recovery counter is not modified from underneath the monitor. A solution may be to disable these portions of the code until the beta-testing terminates.

4.2.2 Rollback-Recovery

Another problem that can be addressed by the monitor is failure-recovery. Low overhead rollback-recovery for long-running applications is an attractive means to add fault tolerance [10]. Such methods, however, have not yet succeeded in industrial settings because it is often difficult to recreate the same execution that happened before the failure and maintain the illusion of continued operation. The monitor and postmortem analysis can be used as a basis for providing an efficient checkpoint/restart facility for the system. Using this technique, the pre-failure execution is replayed exactly. The monitor must be modified to periodically dump the event descriptors on disk and take periodic checkpoints. Also, the network packets can be saved by the local machine instead of relying on a network snooper. This work could be considered as an extension of the PI's Ph.D. thesis, in which the problem of nondeterminism in fault tolerant systems was handled with limited success [10].

4.2.3 Software-based TMR Systems

Like Rollback-Recovery, TMR systems based on active replication have found a limited success in industry. These systems are typically implemented using custom hardware that ensures that the replicas receive the same interrupts and perceive the same memory and device states. Our monitor can be modified to support capture the nondeterminism that occurs in one master replica of a TMR system and force the same events to occur in the backup machines. Such a system would be equivalent to the implementation of the rollback-recovery part, in the sense that the post-failure execution is fundamentally similar to forcing the active replicas of a backup to follow the same course of events and remain identical to the main replica.

4.2.4 Extensions to Distributed Systems

Monitoring the performance of a distributed application and understanding the bottlenecks are lurking problems that will become more acute as distributed programs become more widespread. The technique proposed here is ideal for attacking this problem. By repeating the network interactions of the communicating processes, performance bottlenecks due to networking can be observed during postmortem analysis. The ideas also extend to distributed debugging and distributed rollback-recovery. Based on the experience gained from implementing the initial phases of the project, the extensions and applicability of these techniques to distributed settings will be investigated.

5 Related Work

The effect of the memory hierarchy on system performance has long been an active area of research [1, 3, 6, 7, 8, 9, 16, 18, 19, 20]. A typical study consists of gathering a trace of memory references trace and feeding it to a simulator. Gathering the trace can be done in hardware [11, 17, 20], in firmware [2], or in software [4, 14]. In hardware techniques, a device is inserted on the bus to monitor the traffic and collect memory references. These techniques must disable on-processor caches during trace collection. In addition, because every memory reference must be stored in the trace, the volume of collected data forces the length of the simulation to be no longer than a handful of seconds at best. Firmware techniques rely on modifying the microcode in the processor to collect the trace information. These techniques cannot work with modern RISC processors, and like hardware techniques, they cannot trace the system for more than a few seconds. Software techniques modify the traced program to generate the trace information without relying on hardware support. These techniques, however, result in large time and memory distortions (or dilations). Chen and Bershad report that by using the *epoxie* annotation tool, the size of the traced program increases by a factor of two, while the execution time increases by more than an order of magnitude [6]. The resulting volume of traced data also forces the tracing to stop each two seconds because of trace buffer fillups. The machine used for the experiment had more than 48 Megabytes allocated for trace buffers [5]. These dilation effects make it very difficult to observe the behavior of the system with a real-time workload, or when network activities are involved. The tool presented in Section 3.3 is projected to have very small time and memory dilations. This technique captures highly accurate traces that are almost impossible to obtain using existing software techniques. Furthermore, because the trace information is small, the projected trace durations are three orders of magnitude longer than is possible today. Real-time and network applications can thus be monitored for extended periods of time, and a production system can be traced without a noticeable effect on performance.

Gray gives a good description of Heizenbugs and an experience of their effects on a production database system [12]. To the best of my knowledge, there is no consensus on how to solve the problem of Heizenbugs in the context of kernel implementations. The frequency of crashes of many commercial and research operating systems are mostly due to these bugs, and show that the problem is yet to be solved.

Process checkpointing, as well as monitoring and replaying nondeterministic events have been commonly used for debugging and fault tolerance [15, 10]. In my thesis, I have shown that the cost of monitoring nondeterministic events is very small, except when asynchronous interrupts are involved. This problem has prevented many log-based rollback-recovery research prototypes from having a positive impact in practice. The use of debugging support available on modern RISC architectures is projected to remove the performance problems of monitoring asynchronous interrupts.

6 Outline of Research Plan

The proposed project is a three-year research effort. It is perhaps relevant here to mention that the PI has gained a good experience while implementing his thesis project. That project included a limited form of nondeterminism tracking and extensive kernel instrumentation. Previous experience in implementing the HA-NFS file server with IBM T.J.Watson also provided the PI with a good insight into the UNIX kernel (see the biographical sketch).

Implementation will take place either in the Mach kernel or in BSD UNIX 4.4. The study required to select either system is currently underway. The work effort will proceed according to the following phases:

Phase 1: **(9 months)** • Implement the basic kernel instrumentation required for the monitor and the initial testbed for saving the machine sate, collecting the information, and the associated storage management.

- Implement the network snooping machine.

Phase 2: (3 months) • Evaluate the monitor, focusing on the execution and storage overheads. Test the performance projections made in Section 3.4.

- Compare the implementation with other software techniques that are widely available to establish the viability of the proposed techniques and quantify their improvement over the existing state of the art.
- Develop a few benchmarks for testing purposes.

Phase 3: (6 months) Implement the postmortem analyzer with its dual mode of operation (single stepping or user controlled). The tool will be designed in such a manner that user routines can be compiled within it inside the kernel if necessary. The appropriate system traps will also be added to communicate with the analyzer from the user level.

Phase 4: (6 months) Perform an a study on the monitored operating system. Studies will include the memory architecture interaction with the network, the behavior of the system's TLB under a realistic workload, and the interactions of different system components. Other studies will be performed as necessary.

Phase 5: (6 months) • Add necessary modifications to the monitor in order to help track intermittent bugs.

- Add analysis routines to the postmortem analyzer to help reasoning about system crashes.
- In parallel with the debugging work, add necessary modification to implement the rollback-recovery part using the monitor for gathering recovery information, and the postmortem analyzer as a controller for re-execution.
- Estimate overheads and evaluate performance of the rollback-recovery implementation.

Phase 6: (6 months) • Investigate the applicability of the techniques to performance monitoring of a distributed system.

- Investigate the applicability also to distributed debugging.
- Perform the final evaluation of the project, and identify directions for follow-up research.

7 Impact of the Proposed Research

Tuning the performance of an operating system and debugging it remain two difficult problems. The work proposed here aims at generating a tool for accurate performance measurements under realistic workloads. Instead of measuring the performance of a particular portion of the system in isolation of other interactions, the proposed tool will allow performance to be studied in a comprehensive manner. Furthermore, the monitoring techniques are applicable to production-use systems which can be very valuable in collecting field information. The new potential advances of the work proposed here are as follows:

- A new monitoring tool will be developed to produce execution traces that are almost free of memory and time distortions. In contrast, existing software techniques for generating execution traces introduce substantial slowdowns which can typically be a factor of 10 at best.
- The new monitoring tool has a small storage overhead which will allow it to collect long traces that are estimated in the ranges of tens of minutes. Existing techniques cannot accumulate traces in excess of four or five seconds because of the large amount of data produced.
- The proposed technique will allow a performance study of operating systems under realistic workloads. The performance study can accommodate real-time workloads and network effects, which are very difficult to study under existing techniques.
- The restrictions imposed by the monitoring software are minimal. In contrast, software techniques require the instrumentation of each working program in the system in addition to the operating system. Hardware techniques, on the other hand, cannot collect trace information unless the internal on-chip caches of the processor are disabled, introducing serious time dilations.
- The methodology used here has the potential to impose some discipline during system's code beta-testing.
- The work has the potential to study the performance of distributed and parallel programs, which constitute the next challenging front.
- The proposed technique can be used to efficiently track nondeterministic execution in a log-based rollback-recovery system. This problem remains the most serious one that prevents these techniques from having an impact in practice.
- The proposed technique can be used to build software-based TMR systems out of off-the-shelf workstations and networks.

References

[1] A. Agarwal, J. Hennessy, and M. Horowitz. Cache performance of operating system and multiprogramming workloads. *ACM Transactions on Computer Systems*, 6(4):393–431, November 1988.

[2] A. Agarwal, R.L. Sites, and M. Horowitz. ATUM: A new technique for capturing address traces using microcode. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 119–127, June 1986.

[3] T.E. Anderson, H.M. Levy, B.N. Bershad, and E.D. Lazowska. The interaction of architecture and operating system design. In *Proceedings of the 4th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 108–120, April 1991.

[4] A. Borg, R.E. Kessler, G. Lazana, and D. Wall. Long address traces from RISC machines: Generation and analysis. Technical Report WRL Research Report 89/14, Digital Equipment Corporation Western Research Laboratory, 1989.

[5] J.B. Chen. Memory behavior for an X11 window system. To appear in the Winter 1994 USENIX Conference, January 1994.

[6] J.B. Chen and B.N. Bershad. The impact of operating system structure on memory system performance. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 120–133, December 1994.

[7] J.B. Chen, A. Borg, and N.P. Jouppi. A simulation based study of tlb performance. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 114–123, May 1992.

[8] D.W. Clark. Cache performance in the VAX-11/780. *ACM Transactions on Computer Systems*, 1(1):24–37, February 1983.

[9] D.W. Clark and J.S. Emer. Performance of the VAX-11/780 translation buffer: Simulation and measurements. *ACM Transactions on Computer Systems*, 3(1):31–62, February 1985.

[10] E.N. Elnozahy. *Manetho: Fault Tolerance in Distributed Systems Using Rollback-Recovery and Process Replication.* PhD thesis, Rice University, October 1993. Also available as technical report TR-93-212.

[11] J.K. Flanagan, B.E. Nelson, J.K. Archibald, and K. Grimsrud. BACH: BYU address collection hardware, the collection of complete traces. Technical Report TR-A150-92.1, Brigham Young University, 1992.

[12] J. Gray. Why do operating systems stop and what can be done about them. Proceedings of the 11th Symposium on Reliable Distributed Systems, 1992 1992. Keynote speech.

[13] Hewlett Packard. *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*, 1991.

[14] J.R. Larus. *Abstract Execution: A Technique for Efficiently Tracing Programs.* PhD thesis, University of Wisconsin-Madison, 1990.

[15] T.J. LeBlanc and J.M. Mellor-Crummey. Debugging parallel programs with Instant Replay. *IEEE Transactions on Computers*, C-36(4):471–482, April 1987.

[16] J.C. Mogul and A. Borg. The effect of context switches on cache performance. In *Proceedings of the 4th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 75–84, April 1991.

[17] D. Nagle, R. Uhlig, and T. Mudge. Monster: A tool for analyzing the interaction between operating systems and computer architectures. Technical Report CSE-TR-147-92, University of Michigan, November 1992.

[18] D. Nagle, R. Uhlig, T. Stanley, S. Sechrest, T. Mudge, and R. Brown. Design tradeoffs for software-managed tlbs. In *Proceedings of the 20Th Annual International Symposium on Computer Architecture*, pages 27–38, May 1993.

[19] J. Ousterhout. Why operating systems aren't getting faster as fast as hardware. In *Proceedings of the Summer 1991 USENIX Conference*, pages 247–256, June 1991.

[20] J. Torrellas, A. Gupta, and J. Hennessy. Characterizing the caching and synchronization performance of a multiprocessor operating system. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 162–174, September 1992.